EXHIBIT F

1. Introduction

ActiveTable™ is a product for enabling data mining and querying on tabular data present in web pages. It is a customizable component that can be easily manipulated using a builder tool, enabling developers to quickly create a customized data querying and mining application for their web pages.

ActiveTable consists of three sub-components :

1. The **FastPattern** component, which enables shows a user some quick context-sensitive results based on elementary data mining on a single data source.
2. The **UserQuery** component, which enables users to ask their own questions, using multiple data sources
3. The **DataMining** component, which enables users to ask complex data mining questions, across multiple data sources.

In accordance with the sub-components mentioned above, ActiveTable™ development will proceed in three stages, viz. :

1. Development of basic ActiveTable components, and all classes required to support the **FastPattern** mechanism
2. Support for the **UserQuery** component, and multiple data sources
3. Support for the **DataMining** component

This document addresses the design for the first stage of ActiveTable development, viz. Basic ActiveTable components, and **FastPattern** support.

Section 2 deals with the ActiveTable architecture, and discusses the component classes in detail.

1.1. Document Conventions

Throughout this document, Class and Interface names will be in **bold**. Methods and properties of a class will be in ***bold italics***. Other variables will be in *italics*.

2. Components of ActiveTable™

ActiveTable will be implemented as a JavaBean, that can be visually manipulated and customized using a builder tool.

2.1. ActiveTable : Basic Functionality

ActiveTable requires some basic functionality to be put in place, before further additions (such as the FastPattern mechanism) can be addressed. This includes methods for creating and customizing the ActiveTable. This section deals with all the classes required for supporting this basic functionality.

2.1.1. The ActiveTable Class

The **ActiveTable** class is the basic class required for implementing an ActiveTable. This class extends the **Applet** class, and is a JavaBean. In the first stage of the ActiveTable implementation, this is the only JavaBean that is exposed to the developer.

The **ActiveTable** class contains all information related to the data in the table (including indexing mechanisms for fast data retrieval), as well as information about the look and feel of the data on the user's screen.

In addition, for FastPattern implementation, the ActiveTable class contains information related to the analyses to be performed for the user.

An ActiveTable must have a Primary Data Source. This is the source of the data displayed on the user's screen. In addition, it could have one or more Secondary Data Sources. These are the sources for data that are pulled in when the user mines data across different data sources. Only Primary Data Sources are supported in this version of ActiveTable. Data from the Primary Data Source are imported into the ActiveTable when it is created by the developer. An indexing mechanism (using the **Index** class) is then created for this table.

An ActiveTable also contains a two-dimensional array of **GridElements**. These represent the cells of the table displayed by the ActiveTable.

2.1.1.1 Constructors

2.1.1.1.1 Public ***ActiveTable*** ()

This constructor creates a new **ActiveTable** object with all properties set to NULL.

2.1.1.1.2 Public **ActiveTable** (**DataSourceTypeEnum** *newDataSourceType*, String *newPrimaryDataSource*)

This constructor creates a new **ActiveTable** object with the ***primaryDataSourceType*** property set to newDataSourceType, and the **primaryDataSource** property set to *newPrimaryDataSource*. It also does the following :

1. Creates a new instance of a **DataSource** class by importing data from *newPrimaryDataSource*
2. Creates ***tabularData [ ]*** (an array of objects of class **GridElement**), using the **DataSource** object created above, and default values for grid size, font weight, font size and color.
3. Destroys the **DataSource** object created above.

2.1.1.1.3 Public ***ActiveTable*** (**DataSourceTypeEnum** *newDataSourceType*, String *newPrimaryDataSource*, **AttributeCluster** ***newFastPatternAnalyses[ ]***)

This constructor creates a new **ActiveTable** object with the ***primaryDataSourceType*** property set to newDataSourceType, the **primaryDataSource** property set to *newPrimaryDataSource*, and the ***fastPatternAnalyses[ ]*** property set to *newFastPatternAnalyses[ ]*. It also does the following :

1. Creates a new instance of a **DataSource** class by importing data from *newPrimaryDataSource*
2. Creates ***tabularData [ ]*** (an array of objects of class **GridElement**), using the **DataSource** object created above, and default values for grid size, font weight, font size and color.
3. Creates ***tableIndex***, an index of the values appearing in each of the **AttributeClusters** in the ***fastPatternAnalyses*** array, and ***fastPatternTotals***, an array containing totals of the columns in the ***fastPatternAnalyses*** array (this is concurrent with Step 2, to avoid multiple passes on the data)
4. Destroys the **DataSource** object created above.

2.1.1.2 Properties

2.1.1.2.1 Private **DataSourceTypeEnum** ***primaryDataSourceType***

PrimaryDataSourceType contains the Primary Data Source Type (ASCII / ODBC / HTML)

2.1.1.2.2 Private String ***primaryDataSource***

primaryDataSource is a string that contains the name of the Primary Data Source of the ActiveTable

2.1.1.2.3 Private **GridElement** ***tabularData[ ]***

TabularData is a two-dimensional array of **GridElements**, representing the actual table displayed.

2.1.1.2.4 Private Dimension ***defaultGridSize***

defaultGridSize contains the default grid size for the grid elements.

2.1.1.2.5 Private int ***defaultFontWeight***

DefaultFontWeight contains the default font weight of the text displayed in the grid element.

2.1.1.2.6 Private int ***defaultFontSize***

DefaultFontSize contains the default font size for the text displayed in the grid element.

2.1.1.2.7 Private Color ***defaultFontColor***

DefaultFontColor contains the default font color of the text displayed in the grid element.

2.1.1.2.8 Private Index ***tableIndex***

TableIndex contains indexing information about the table displayed by the ActiveTable.

2.1.1.2.9 Private AttributeCluster ***fastPatternAnalyses[ ]***

FastPatternAnalyses is an array of AttributeClusters that facilitate FastPattern analysis.

2.1.1.2.10 Private ColumnInfo ***fastPatternTotals [ ]***

FastPatternTotals is an array containing information about the totals for various columns appearing in the ***fastPatternAnalyses*** array.

2.1.1.2.11 Private InfoBalloon ***fastPatternBalloon***

FastPatternBalloon contains the FastPattern information to be displayed to the user when she moves her mouse around on the ActiveTable.

2.1.1.2.12 Private FIFO ***fastPatternCache***

FastPatternCache contains a cache of the previous FastPatterns viewed by the user.

2.1.1.3 Methods

2.1.1.3.1 Public void ***setPrimaryDataSourceType*** (DataSourceTypeEnum *newPrimaryDataSourceType*)

Sets ***primaryDataSourceType*** to *newPrimaryDataSourceType*.

2.1.1.3.2 Public DataSourceTypeEnum ***getPrimaryDataSourceType*** ()

Returns ***dataSourceType***

2.1.1.3.3 Public void ***setPrimaryDataSource*** (String *newPrimaryDataSource*)

Sets ***primaryDataSource*** to *newPrimaryDataSource*.

2.1.1.3.4 Public String ***getPrimaryDataSource*** ()

Returns ***primaryDataSource***.

2.1.1.3.5

The various elements of the ActiveTable Applet are explained below :

1. ActiveTable Applet
2. GridElement
3. ActiveTable
4. Text Area
5. Buttons

2.3. GridElement (extends TextArea)

A GridElement is an object that contains a single element of the data to be displayed and analyzed. It is not a JavaBean, and is not directly available for manipulation by the developer, but serves as a building block for an ActiveTable.

2.3.1. Constructors

2.3.1.1 Public Void GridElement()
Constructs a "floating" GridElement object of default size with data = 0.0.

2.3.1.2 Public Void GridElement(Int row, int col, float data)
Constructs a GridElement object of default size anchored to the specified row and column, with data as specified.

2.3.1.3 Public Void GridElement (int row, int col, Dimension Size)
Constructs a GridElement object of the specified size, anchored to the specified row and column.

2.3.1.4 Public Void GridElement (int row, int col, Dimension Size, float data)
Constructs a GridElement object of the specified size, anchored to the specified row and column, and containing the data specified.

2.3.2. *Properties*

2.3.2.1 Private int row
Specifies the row of the ActiveTable to which the GridElement is anchored.

2.3.2.2 Private int column
Specifies the column of the ActiveTable to which the GridElement is anchored.

2.3.2.3 Private Dimension size
Specifies the size of the GridElement.

2.3.2.4 Private boolean numeric
Specifies whether the GridElement contains numeric data. If this element is true, the GridElement contains numeric data. If it is false, it contains string (textual) data.

2.3.2.5 Private float data
Specifies the data contained in the GridElement. (if isNumeric is true)

2.3.2.6 Private String displayText
Specifies the text displayed in the GridElement when it appears in an ActiveTable. DisplayText always contains the string representation of the data contained in the GridElement.

2.3.2.7 Private int fontSize
Specifies the font size for the GridElement

2.3.2.8 Private int fontWeight
Specifies the font weight for the GridElement

2.3.2.9 Private boolean bold
Specifies whether the GridElement contents are to be displayed in **bold** font.

2.3.2.10 Private boolean italicized
Specifies whether the GridElement contents are to be displayed in *italics*.

2.3.2.11 Private Color color
Specifies the color of the GridElement contents

2.3.3. *Methods*

2.3.3.1 Public int getRow ()
Returns the row (value of the *row* property) to which the GridElement is anchored.

2.3.3.2 Public void setRow (int newRow)
Sets the row to which the GridElement is anchored (sets the *row* property to newRow)

2.3.3.3 Public int getColumn ()
Returns the column (value of the *column* property) to which the GridElement is anchored.

2.3.3.4 Public void setColumn (int newColumn)
Sets the column to which the GridElement is anchored (sets the *column* property to newColumn)

2.3.3.5 Public Dimension getSize ()
Returns the size (value of the *size* property) of the GridElement.

2.3.3.6 Public void setSize (Dimension newSize)
Sets the size of the GridElement. (Sets the *size* property to newSize)

2.3.3.7 Public boolean isNumeric ()
Returns information about whether the GridElement contents are numeric (returns the value of the *numeric* property)

2.3.3.8 Public void setNumeric (boolean newNumeric)
Sets the *numeric* property of the GridElement to newNumeric.

2.3.3.9 Public float getData ()
Returns the data contained in the GridElement (value of the *data* property).

2.3.3.10 Public void setData (float newData)
Sets the data contained in the GridElement. (sets the *data* property to newData).

2.3.3.11 Public String getDisplayText ()
Returns the text displayed by the GridElement (value of the *displayText* property)

2.3.3.12 Public void setDisplayText (String newDisplayText)
Sets the text displayed by the GridElement (sets the *displayText* property to newDisplayText).

2.3.3.13 Public int getFontSize ()
Returns the font size of the GridElement (value of the fontSize property).

2.3.3.14 Public void setFontSize (int newFontSize)
Sets the font size of the GridElement. (Sets the *fontSize* property to newFontSize).

2.3.3.15 Public Int getFontWeight ()

Returns the font weight of the GridElement. (value of the *fontWeight*).

2.3.3.16 Public void setFontWeight (int newFontWeight)

Sets the font weight of the GridElement. (sets the *fontWeight* property to *newFontWeight*).

2.3.3.17 Public boolean isBold ()

Returns information about whether the text in the GridElement is displayed in bold letters. (value of the *bold* property).

2.3.3.18 Public void setBold (boolean newBold)

Sets the *bold* property of the GridElement to newBold.

2.3.3.19 Public boolean isItalicized ()

Returns information about whether the text in the GridElement is italicized. (value of the *italicized* property).

2.3.3.20 Public boolean setItalicized (boolean newItalicized)

Sets the *italicized* property of the GridElement to newItalicized.

2.3.3.21 Public Color getColor ()

Returns the current color of the text displayed in the GridElement (value of the *color* prperty)

2.3.3.22 Public void setColor (Color newColor)

Sets the color of the GridElement (sets the *color* property to newColor).

2.3.3.23 Public void displayBalloon (Balloon newBalloon)

Displays newBalloon at the GridElement's lower right corner.

2.3.4. Events

2.4. ActiveTable
2.5. Text Area
2.6. Buttons

An ActiveTable can have one or more analyses associated with it. These are the entities that control the nature of the pattern displayed to the user for FastPattern implementation. They are represented using the **AttributeCluster** class.

Issues in Data Mining on the Internet

The following issues are considered in this document :

1. Consumer vs. Corporate user

This is more of a "thin client" vs. "Fat Client" issue.
"Thin Client" : as memory constraints are likely to exist, it would be preferable to mine the data on the remote server, and bring the results across to the client.
"Fat Client" : it would be useful to take advantage of the computational power of the client, and mine the data on the client.

Scenario	Pros	Cons
Data Mining on the client	1. No special server software required to process Data Mining requests 2. Web / Database Server is not loaded 3. No extra traffic on the network	1. Client may take longer to show Data Mining results
Data Mining on the server	1. Special server software required 2. Extra load on Web / Database server 3. Server software repeats work for same Data Mining queries from different clients 4. Extra traffic on network, to transmit results	1. Server is scaleable, and hence, *might* be faster

Note . It may not always be possible for a designer to tell beforehand whether the client is likely to be "thin" or "fat" On the Internet, this is practically impossible. On a corporate Intranet, however, this might be possible.

2. Runtime (user) vs. Design-time (developer) mining

Design-time Mining : Using Java Beans, it is possible to save the state of various objects at the time of design. This implies that when a developer designs an ActiveTable™, she can specify the Data Mining queries, and these could be saved along with the object. When the client requests the object, the results of the queries are sent along with the object.
Runtime Mining : Only the data mining code and data are sent to the client, and the actual mining is performed there.

Scenario	Pros	Cons
Design-time Mining	1. User does not have to wait for mining results 2. The (presumably higher) computational power of the server can be used.	1. Extra traffic on the network (as the Data Mining results have to be sent along with the applet) 2. The results sent to the client may not ever be seen by the user
Runtime Mining	1. No extra load on the network (as the client does her own mining) 2. Client mines data only when required, so all results are always seen by the user	1. User may have a long wait for mining results (especially for "thin" clients)

3. **Small tables vs. large tables** .

The "smallness" of a table, with regard to its amenability for Data Mining, is determined by the following factors:

a. The number of attributes selected by the user for data mining – this can be limited to 3 or 4
b. The number of unique values for each attribute chosen
c. The depth of mining required – this can be limited
d. The number of records to be mined.

Mining is faster on small tables than on large tables.
The table "smallness", more than any of the above factors, should guide the decision of where the data mining is performed. Consider :

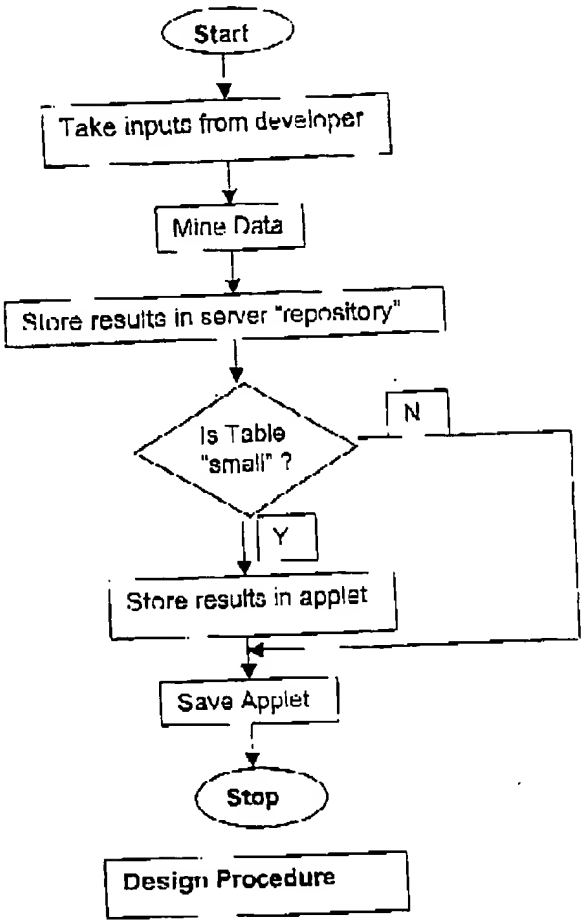
a. It is not always possible to determine whether a potential client is "thin" or "fat". However, it is **always** possible to determine the "smallness" of a table, at the time when the ActiveTable™ is being designed.
b. The ability of the client to mine data in as short a time as possible is obviously related to the "smallness" of the table being mined. If the table is small enough, even an "ultra-thin" client may be able to mine satisfactorily, and if the table is huge, even a big server may take minutes to mine it

4 ActiveX vs. Java

We are dependent on the Java-to-ActiveX Bridge to convert code from one model to the other. This has to be tested thoroughly to ensure that the conversion is accurate. If necssary, code modifications may have to be made.

An alternative design paradigm
The design methodology suggested below avoids some of the disadvantages of mining data purely on the server, or purely on the client.
The design algorithm would be as follows :

1. Based on the table size, and the developer inputs (when the ActiveTable™ is being designed), decide whether the table is "small" or "big".
2. For both types of tables, mine the data on the server, during the design phase. (The possibility of batch-processing several ActiveTable™ designs, so as to obviate the need for the designer's presence during the mining can be investigated).
3. For both types of tables, store the Data Mining results on a "repository" on the server. This "repository" is an intelligent entity, that can respond to RMI (Remote Method Invocations) calls by clients, responding with Data Mining results, as and when required.
4. For "small" tables, store the results along with the applet also. If, for some reason, the client is not able to load all the results, the "repository" would serve as a backup.
5. For "big" tables, do not store the results on the client applet.
6. When the applet is sent to the client, and the user requests some mining results, the applet first checks its local store. If the result is found, it is displayed to the user. If it is not, the client invokes a method on the remote "repository" entity, and displays the results returned.


Start
Take inputs from developer
Mine Data
Store results in server "repository"
Is Table "small" ?
N
Y
Store results in applet
Save Applet
Stop
Design Procedure

This method has significant advantages :

1. For small tables, access to Data Mining results will be extremely fast.
2. For big tables, the data have already been mined. Only the results are requested from the server. Consequently, network load is reduced, and access time is lower.
3. For big tables, the data mining results are sent to the client only when the user asks for them. Consequently, the problem of sending extra information to each client is avoided. For small tables, all the results are still sent to the client, whether they are displayed to the user or not, but this is probably less of a problem, due to the "smallness" of the table.
4. The server will never be faced with a scenario where it is mining the same data several times to answer the same question from several clients. Since the results are stored, it will merely have to fetch the results and transmit them to the client.
5. It is possible to cache results on the client, even for big tables, so that repeated requests do not have to be made to the server. This speeds up access even further.